Universitat Autònoma de Barcelona

escola d'enginyeria

5731 - Anàlisi de les claus criptogràfiques utilitzades en Bitcoin

Memòria del projecte final de carrera d'Enginyeria Informàtica realitzat per Josep Miquel Andreu i dirigit per Jordi Herrera.

Bellaterra, Juny de 2014

El firmant, Jordi Herrera i Joancomartí, professor del Departament d'Enginyeria de la Informació i de les Comunicacions de la Universitat Autònoma de Barcelona

CERTIFICA:

Que la present memòria ha sigut realitzada sota la seva direcció per Josep Miquel Andreu Alemany

Bellaterra, juny de 2014

Firmat: Jordi Herrera i Joancomartí

Als meus companys,
que iniciaren amb mi aquest viatge fa 6 anys

Agraïments

A la meva família, per tots els esforços que han fet per a que jo pugui estar escrivint aquest projecte.
Al meu tutor, per introduir-me i guiar-me dins l'increïble món dels bitcoins. I al Joan Melià, per executar els tests del NIST presentats en aquest projecte.

Índex

Índex de figures

Índex de taules

Capítol 1

Introducció

El Bitcoin és una moneda virtual que ha anat guanyant molta popularitat en els últims temps. Al no dependre de cap supervisor per a realitzar transaccions, obre un nou paradigma en l'ús de divises dins Internet ja que fins ara, totes les transaccions i pagaments d'una moneda necessitaven d'un intermediari per a realitzar-se.

Tota la tecnologia i coneixements que hi ha al darrere la fan molt atractiva per poder analitzar-ne el funcionament i entendre'l. El problema principal per a realitzar aquest anàlisi és la magnitud del projecte. En aquest projecte, ens centrarem només en l'àrea de l'ús d'eines criptogràfiques que fa aquesta moneda. Concretament, com realitza la generació de claus públiques i privades.

1.1 Objectius

La línia de treball principal del projecte és analitzar una part del funcionament dels bitcoins: l'ús de claus criptogràfiques i la importància de la seva aleatorietat. Per arribar aquí, són necessaris uns objectius previs per comprendre com funcionen els bitcoins i la criptografia que s'utilitza en ells.

Per concretar la línia de treball, es plantegen els següents objectius:

1. Estudiar les corbes el·líptiques i del seu ús en la criptografia.

2. Introduir el funcionament dels bitcoins.

3. Estudiar la importància de l'ús de generadors de nombres aleatoris que funcionin correctament.

4. Analitzar els recursos que utilitzen els desenvolupadors de bitcoin per a la generació de claus.

5. Testejar el funcionament dels generadors de claus.

1.2 Estructura del document

La memòria presenta cinc capítols més i un apèndix que s'estructuren en:

- **Capítol 2. Fonaments teòrics**: Es comença fent una introducció a les corbes el·líptiques i els seus usos en criptografia. Seguidament s'introdueix el Bitcoin i la necessitat de les corbes el·líptiques.

- **Capítol 3. Anàlisi de codi**: S'analitza el codi encarregat de generar nombres aleatoris de dues wallets Bitcoin: la PyWallet i la BitcoinQt. La primera és interessant per la simplicitat del codi. La segona per ser el client de referència.

- **Capítol 4. Comprovacions**: Es verifica si els usuaris de Bitcoin utilitzen claus aleatòries analitzant un conjunt de clau privades determinat. Després, usant els generadors de nombres aleatoris descrits al capítol anterior, s'analitzen amb els tests d'aleatorietat del NIST.

- **Capítol 5. Deterministic wallets**: Introdueix en un altre tipus de wallets per determinar-ne la generació de claus a nivell teòric.

- **Capítol 6. Conclusions**: Es resumeixen els resultats de l'anàlisi obtinguts dins del projecte. Es proposen noves línies a seguir dins l'àmbit de generació de claus dels Bitcoin.

- **Annexe A. Compilació, instal·lació i execució dels generadors de claus**: S'afegeixen els scripts necessaris per executar els generadors de nombres aleatoris que incorporen els sistemes operatius. A més s'explica com compilar i instal·lar l'última versió d'OpenSSL en entorns Unix.

1.3 Requeriments, recursos i viabilitat

Al ser un projecte d'anàlisi i validació de l'aleatorietat dels nombres, no hi ha uns requeriments determinats. No hi ha intenció d'obtenir uns resultats concrets, sinó de

comprendre com s'implementen les característiques referents a les claus utilitzades en Bitcoin.

Els recursos utilitzats per a l'anàlisi són:

- Ordinadors personals: la feina s'ha elaborat majoritàriament amb un MacBook Pro. Ha estat necessari l'ús de dos ordinadors més amb els sistemes operatius Windows i GNU/Linux per poder obtenir unes mostres de bytes generats aleatòriament amb les llibreries pròpies del sistema.
- Intèrpret de Pyton 2.7: s'han realitzat diferents scripts en aquest llenguatge.
- Compilador gcc: s'ha usat el compilador per executar la funció RAND_bytes de les llibreries OpenSSL.
- SAGE: és un software matemàtic basat en el llenguatge Python on se li han incorporat moltes funcions matemàtiques de sèrie. L'ús d'aquest software facilita molt treballar amb nombres grans i amb les corbes el·líptiques. És totalment OpenSource.
- Llibreries OpenSSL: durant l'anàlisi hi ha hagut la necessitar de compilar i executar la funció RAND_bytes. Les llibreries són OpenSource.

Tots els recursos utilitzats són fàcilment adquiribles i, a excepció dels ordinadors personals, el seu cost és nul. Per tant, el projecte ha estat viable realitzar-lo sense necessitat d'adquirir hardware o software nou.

1.4 Planificació temporal

El projecte està pensat per fer-ne un anàlisi previ durant el primer semestre del curs per poder-ne descriure l'estat de l'art. Tota la part d'anàlisi i comprovacions s'ha realitzat durant el segon semestre del curs.

Un anàlisi d'aquestes característiques és molt difícil de planificar, ja que a priori hom no sap que es pot trobar. La planificació inicial realitzada a l'informe previ recollia una part de recerca sobre els bitcoins i les corbes el·líptiques. Seguidament, realitzar una anàlisi del codi del client de referència per determinar-ne el generador de nombres aleatoris i comparar-lo amb altres clients.

Finalment la programació ha variat substancialment després de veure la complexitat que requeria analitzar el codi del client oficial. Primer s'ha optat per analitzar la

PyWallet, una wallet amb un codi molt intuïtiu que està escrit en Python. Després això ha facilitat la tasca d'anàlisi del client oficial.

Un cop completat l'anàlisi, hi ha hagut temps per a realitzar diverses proves amb els generadors del nombres aleatoris, a més d'una prova d'analitzat un conjunt d'adreces bitcoin per determinar-ne si s'havien utilitzat o no.

Per últim, s'ha pogut incloure una introducció a les Deterministic wallet per a entendre el seu funcionament i comprendre la importància que tenen en la generació de claus.

El diagrama de Gannt final del projecte ha estat el següent:

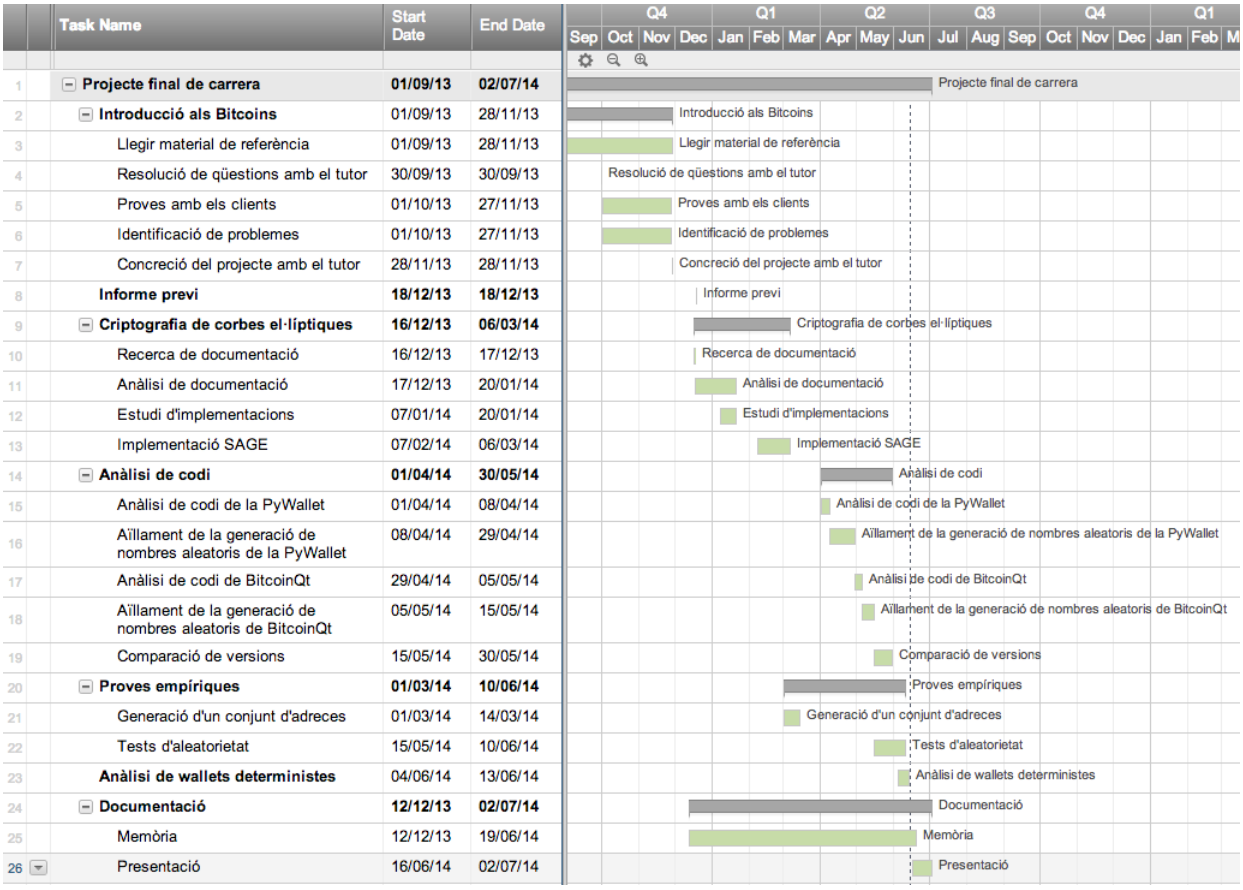


Figura 1.1: Diagrama de Gannt

Capítol 2

Fonaments teòrics

En aquest capítol s'introduirà el concepte de corba el·líptica i el seu ús en la criptografia. Seguidament es donarà una introducció al funcionament del bitcoin i la importància que tenen les corbes el·líptiques en la seva generació de claus i adreces.

2.1 Corbes el·líptiques

Una corba el·líptica [10] sobre un cos K és una corba plana no singular (no conté cúspides o punts d'intersecció amb ella mateixa) donada per una equació polinòmica de grau 3.

En cas de que[1] $char(K) \neq 2, 3$, qualsevol equació d'una corba el·líptica, en el pla afí, té la següent forma:

$$y^2 = x^3 + Ax + B$$

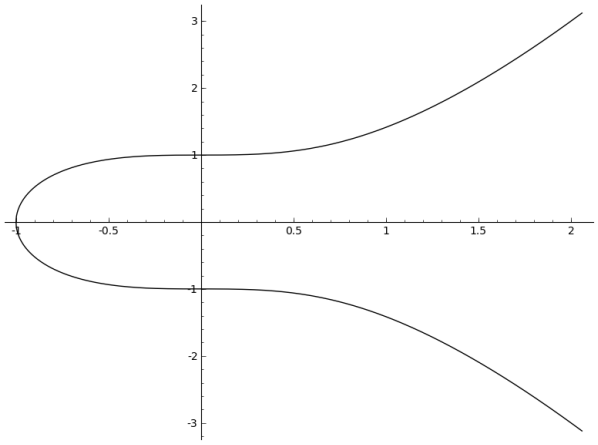
Els punts que pertanyen a la corba s'anomenen punts racionals.

2.1.1 Estructura de grup

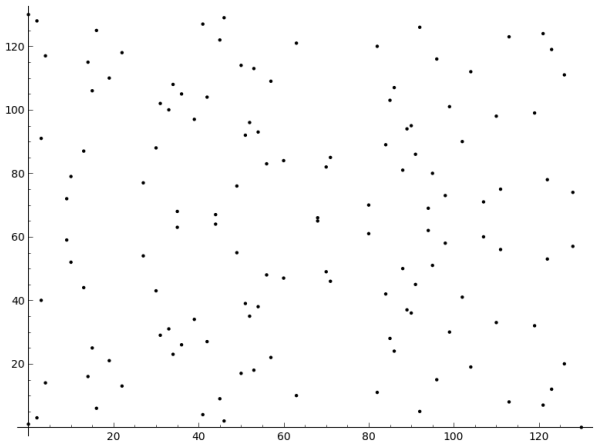
Considerem una corba el·líptica definida en el cos $\mathbb{R}$. Anem a definir una operació per dotar a la corba amb una estructura de grup.

Comencem agafant dos punts de la corba, P i Q i considerem la recta que passa per aquests dos punts. Aquesta recta com a màxim podrà tallar un punt més, anomenem-lo R, que també forma part de la corba. Considerem a més un punt O a

[1] Definim la característica d'un cos K com el nombre enter positiu n més petit tal que $\sum_{i=1}^{n} 1 = 0$ i la denotem per $char(K)$.



(a) Cos $\mathbb{R}$



(b) Cos $\mathbb{F}_{131}$

Figura 2.1: Corba el·líptica $y^2 = x^3 + 1$ en diferents cossos.

l'infinit (és a dir, visualitzem la corba en el pla projectiu). Definim l'operació + de punts d'una corba el·líptica:

- Si una recta uneix els punts P, Q i R, aleshores $P + Q + R = O$.
- L'element neutre de l'operació és O.
- Per tenir una estructura de grup, s'ha de complir que $P + (-P) = O$, on $(-P)$ és l'oposat de P. Aleshores, amb el punt O que hem considerat, si $P = (x, y)$ aleshores $-P = (x, -y)$.

Dels punts anterior, es dedueix que $P + Q = -R$. En la Figura 2.2 podem veure el resultat de la suma del primer exemple:

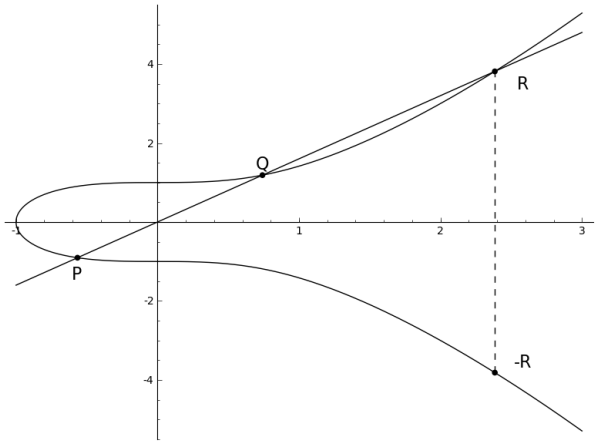


Figura 2.2: Corba $y^2 = x^3 + 1$ i recta $y = 1.6x$. El resultat de $P + Q$ és $-R$

Per a poder mantenir l'estructura de grup, hem de definir els casos en que la recta escollida només talla els dos punts escollits. Això succeeix quan la recta és tangent en un dels dos punts. A més hem de considerar el cas especial d'escollir una recta tangent en un punt que no en talla cap més. Això es resumeix en els següents punts que podem observar gràficament a la Figura 2.3:

- Si la recta és tangent a algun dels dos punts escollits, aquest punt compta dues vegades en l'operació definida. Per exemple, si la recta és tangent a la corba en el punt Q, aleshores els tres punts a considerar són P, Q i Q. Per tant tenim que $P + Q + Q = O$ i $P + Q = -Q$.

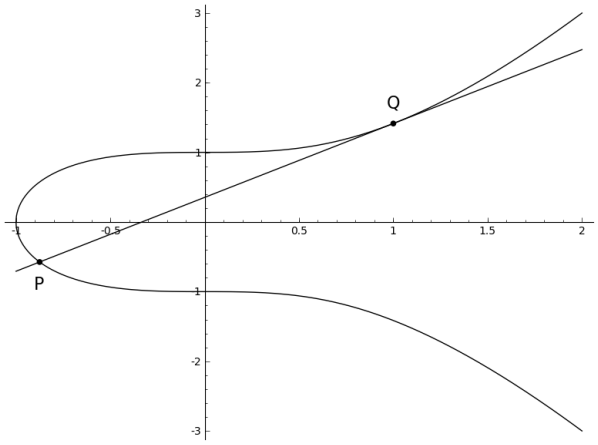
- Si la recta és paral·lela a l'eix y i no és tangent a cap punt de la corba, definim el tercer punt com a O. Per tant, $P + Q + O = O$ i $P + Q = O$.

- Si la recta és paral·lela a l'eix y i alhora tangent a un punt de la corba, només tallarem la corba per un punt. Aquest punt compta dues vegades i definim el tercer punt com a O, seguint les dues imposicions anteriors. Per tant, $P + P + O = O$ i $P + P = O$.

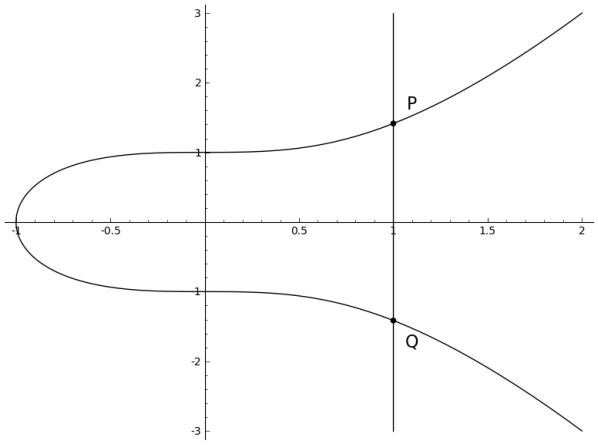
Es pot demostrar que aquesta estructura donada a les corbes el·líptiques genera un grup abelià. Al no ser la finalitat d'aquest projecte, no s'adjunta la demostració.

2.1.2 Corbes el·líptiques en cossos finits

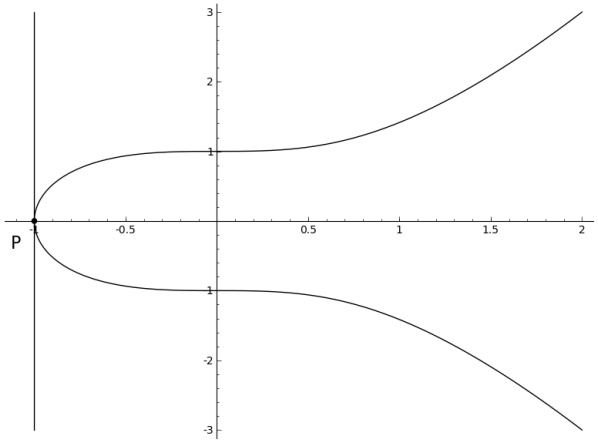
Per dotar de l'estructura de grup a la corba el·líptica, hem treballat geomètricament sobre $\mathbb{R}$. En cas de tenir corbes sobre cossos finits, les corbes passen a ser punts sobre



(a) Recta $y = \frac{3}{2*\sqrt{2}}x + \sqrt{2} - \frac{3}{2\sqrt{2}}$, que és tangent al punt Q



(b) La recta és paral·lela a l'eix y però no és tangent als punts P i Q



(c) La recta és paral·lela a l'eix y i tangent al punt P

Figura 2.3: Situacions especials de les rectes sobre la corba $y^2 = x^3 + 1$

el pla i es perd la visió geomètrica. Per aquest motiu, expressarem analíticament el càlcul de l'operació +, de forma que segueixi essent coherent en els cossos finits.

Considerem una corba el·líptica sobre un cos K amb $char(K) \neq 2, 3$ amb equació $y^2 = x^3 - px - q$. A més, considerem dos punts $P = (x_P, y_P)$ i $Q = (x_Q, y_Q)$. Assumim que $P + Q = -R$ i $R = (x_R, y_R)$.

En primer lloc, suposem que $x_P \neq x_Q$. Definim $s = \frac{y_P - y_Q}{x_P - x_Q}$ com el pendent de la recta que uneix els puns P i Q. Al ser K un cos, aquesta operació està ben definida.

Per tant, la recta que uneix els dos punts tindrà la forma $y = sx + n$. Calculem el valor de n:

$$y_P = sx + n \Rightarrow n = y_P - sx_P$$

Aleshores, la recta té per equació $y = sx + y_P - sx_P$. Per trobar els valors (x_R, y_R) hem de resoldre el sistema d'equacions

$$\begin{cases} y^2 &= x^3 - px - q \\ y &= sx + n \end{cases}$$

Substituint y a la segona equació tenim

$$(sx + n)^2 = x^3 - px - q$$

Desenvolupant l'equació, arribem a

$$x^3 - s^2x^2 - (p + 2sn)x - q - n^2 = 0$$

Però a més, sabem que x_P i x_Q són solucions d'aquesta equació. Aleshores, a l'haver suposat cossos amb $char(K) \neq 2, 3$ tenim que

$$x^3 - s^2x^2 - (p + 2sn)x - q - n^2 = (x - x_P)(x - x_Q)(x - x_R)$$

i podem obtenir el valor[2] d'x_R mitjançant el terme de grau 2:

$$-s^2 = -x_P - x_Q - x_R \Rightarrow x_R = s^2 - x_P - x_Q$$

Ara fàcilment podem trobar el valor de la y_R substituint a l'altra equació:

$$y_R = sx_R + n = sx_R + y_P - sx_P = y_P + s(x_P - x_R)$$

[2]Recordem que $(x - a)(x - b)(x - c) = x^3 - (a + b + c)x^2 + (ab + ac + bc)x - abc$

Per tant, tenim que

$$P + Q = -R = (x_R, -y_R) = (s^2 - x_P - x_Q, -y_P + s(x_R - x_P))$$

Suposem ara que $x_P = x_Q$. Ens podem trobar en tres casos.

- Si $y_P = -y_Q$, definim $P + Q = O$, de manera que que $P = -Q = -(-P)$ com havíem definit a l'apartat anterior.
- Si $y_P = y_Q \neq 0$, tenim $P = Q$ amb un pendent diferent de 0. Mitjançant la derivada de la corba, obtenim que:

  $$s = \frac{3x_P^2 - p}{2\sqrt{x_P^3 - px_P - q}} = \frac{3x_P^2 - p}{2y_P}$$

  i les equacions queden simplificades com:

  $$\begin{cases} x_R &= s^2 - 2x_P \\ y_R &= y_P + s(x_P - x_R) \end{cases}$$

  i per tant, $P + P = 2P = -R = (s^2 - 2x_P, -y_P + s(x_P - x_R))$
- Si $y_P = y_Q = 0$, $P + P = O$, ja que tenim una recta amb pendent infinit.

Amb aquesta formalització analítica, podem usar els resultats obtinguts per treballar en cossos finits. L'única condició és que aquests cossos tinguin $char(K) \neq 2, 3$.

2.2 Criptografia de corbes el·líptiques

La criptografia de corbes el·líptiques és una variant de la criptografia asimètrica. Es basa en l'ús de les corbes el·líptiques per obtenir les claus pública i privada. L'avantatge principal del seu ús respecte altres criptosistemes com l'RSA, és que per obtenir un nivell de seguretat equivalent no és necessari tenir una longitud de clau tan gran.

A l'igual que el logaritme discret, amb les corbes el·líptiques i l'operació de suma de punts donada, sorgeix un problema de molta complexitat. Donat un punt inicial P d'ordre n d'una corba, és molt senzill calcular kP, on k és un nombre de l'interval $[1, n-1]$. En canvi, calcular k a partir del resultat kP coneixent P és un problema d'una complexitat molt gran per cossos finits amb molts elements.

Així doncs, donat un cos finit i una equació polinòmica de grau 3, podem construir una corba el·líptica per a després utilitzar-la en criptografia. Tot i així, hi ha una sèrie de corbes que no es consideren adequades pel seu ús en criptografia. Un exemple són les corbes supersingulars i les corbes anòmales.

Per definir les corbes supersingulars, necessitem una aplicació d'un teorema, anomenat "Teorema de Hasse sobre corbes el·líptiques". Els resultats del teorema s'escapen de l'abast d'aquest projecte per la seva complexitat, però hi ha un resultat senzill que fa referència al nombre d'elements d'una corba el·líptica. És clar que en un cos finit tenim un nombre finit d'elements, però no tots aquests elements han de pertànyer a la corba. Aleshores, el Teorema de Hasse ens diu que el nombre d'elements de la corba Ω sobre el cos $\mathbb{F}_q$ compleix:

$$\sharp\Omega = q + 1 - t$$

on $|t| \leq 2\sqrt{q}$. En el cas de tenir una corba amb $t \equiv 0 \mod q$, diem que aquesta corba és supersingular.

El concepte de corba anòmala és més senzill. Si el nombre d'elements d'una corba Ω sobre un cos $\mathbb{F}_q$ és q, aleshores les corba és anòmala.

Existeixen algorismes eficients per a aquest tipus de corbes que ens permeten trencar els criptosistemes de clau pública que en fan ús, ja que amb la clau pública se'n pot obtenir la privada. Per aquest motiu, no és recomenable el seu ús en criptosistemes que funcionin amb corbes el·líptiques.

2.2.1 Generació de claus de l'ECDSA

DSA són les sigles de *Digital Signature Algorithm*, un algorisme estàndard del Govern Federal dels Estats Units d'Amèrica per l'ús de signatures digitals. És una variant de l'algorisme ElGamal i a l'igual que aquest, és un algorisme de clau pública. Ambdós es basen en la complexitat de resoldre el problema del logaritme discret[3].

L'algorisme DSA ha estat adaptat per usar les corbes el·líptiques en lloc d'usar els logaritmes discrets. És el que es coneix com *Elliptic Curve Digital Signature Algotithm* (ECDSA)

Seguidament, es detalla com es generen les claus públiques per l'ECDSA:

[3]A diferència del càlcul del logaritme en $\mathbb{R}$, no és senzill el càlcul del logaritme en cosos finits. En canvi, si que és senzill calcular l'operació inversa: l'exponenciació.

1. Es selecciona una corba el·líptica Ω i un punt P d'ordre n que pertanyi a la corba.
2. Seleccionem aleatòriament un nombre k de l'interval $[1, n-1]$.
3. Calculem $Q = kP$.
4. k és la clau privada i el punt Q la clau pública.

Les claus obtingudes permeten elaborar una signatura digital i verificar-la. El mètode amb el qual generem les claus en aquest algorisme és molt important, ja que s'usa en la generació d'adreces i claus dels Bitcoins, com es veurà més endavant.

No s'entra en detall pel que fa el procés de signatura i verificació de l'algorisme ECDSA ja que no és rellevant per al desenvolupament d'aquest projecte.

2.2.2 Corba el·líptica SECP256K1

La corba el·líptica que porta el nom SECP256K1 és la que utilitzen els bitcoions internament, juntament amb l'algorisme ECDSA. Els seus paràmetres són recomanats per l'empresa Certicom, subsidiària de BlackBerry. Prenent com a equació model de la corba el·líptica $y^2 = x^3 + Ax + B$, els paràmetres són:

- Cos finit: F_p, on $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 24 - 1$
- $A = 0$
- $B = 7$

Pel punt inicial, hi ha dues maneres de descriure'l: amb compresió o sense compresió. La compresió consisteix en només donar la coordenada x del punt, ja que la y es pot calcular fàcilment amb l'equació de la corba. En ambdós casos, s'utilitza un codi de dos dígits a l'inici del punt per diferenciar-los. El codi 04 s'utilitza per a la versió sense comprimir i el 02 per a la versió comprimida.

En versió comprimida, el punt inicial és:

G=02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798

En versió no comprimida és:

G=04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8

2.3 Bitcoins

El bitcoin és una moneda electrònica descentralitzada que entrà en funcionament l'any 2009 [4]. Fou concebuda per una persona (o grup de persones) amb el pseudònim de Satoshi Nakamoto. La seva independència respecte els supervisors bancaris, han fet que sigui una moneda molt popular. Per contra, la manca de supervisors provoca que s'especuli amb ella, fent que el seu valor fluctuï constantment.

2.3.1 Funcionament

El funcionament del bitcoin es basa en dos conceptes claus: les transaccions i els blocs de transaccions. Per entendre'ls, hem de pensar el bitcoin com una xarxa amb diferents nodes que interactuen entre sí.

Les transaccions, equivalentment a les transaccions bancàries habituals, ens permeten enviar bitcoins entre dos usuaris de la xarxa. Cada transacció conté dues parts essencials: les entrades i les sortides. Mitjançant uns indentificadors, ens especifiquen l'origen dels bitcoins (entrades) i el seus destins (sortides). Aquests identificadors actuen de manera similar als nombres de comptes bancaris. Cada transacció està identificada amb un hash obtingut de les entrades i sortides. Aquest hash alhora està signat digitalment per garantir la propietat dels diners transferits. Al següent apartat s'explicarà amb més detall el funcionament d'aquestes adreces.

Totes les transaccions fetes entre els usuaris de la xarxa bitcoin queden agrupades en blocs. Cada bloc que es genera amb les adreces, és verificat i enllaçat amb el bloc anterior que s'ha generat. Aquest sistema permet tenir una cadena de blocs que reflexa tot l'historial de transaccions de la xarxa bitcoin.

Els usuaris encarregats de generar aquests blocs es coneixen amb el nom de "miners". Aquests verifiquen les transaccions realitzades, de manera que els usuaris no gastin dos cops els mateixos bitcoins. Si una transacció no està inclosa dins d'un bloc, aquesta és considerada invàlida.

Els miners estan agrupats en diferents nodes dins tota la xarxa bitcoin. Quan un dels nodes genera un bloc, aquest bloc es propaga per la resta de nodes, garantint que tothom pugui visualitzar les transaccions realitzades. Cada node revisa que el bloc estigui ben generat. Un cop està verificat, tots els nodes miners comencen a generar el següent bloc.

El protocol bitcoin contempla la possibilitat que en un moment donat es generi més

d'un bloc alhora. Això suposa un problema, ja que tenir més d'una branca a la cadena de blocs pot provocar, entre altres problemes, que usuaris puguin gastar-se dos cops els mateixos bitcoins. El que es fa en aquests casos és esperar a que una de les dues branques de la cadena s'allargui més que l'altra. Quan això succeeix, s'ignora l'altra branca. Per aquest motiu, sempre es recomana que les transferències es considerin permanents quan s'han generat uns 6 blocs per sobre d'elles.

Ja hem vist com podem fer transaccions entre comptes bitcoins i com es validen. Però com es creen i guarden els bitcoins? En el sistema bancari, són els bancs centrals els encarregats d'emetre una moneda. Aquesta moneda a més, té un suport físic en forma de monedes i bitllets.

En el cas dels bitcoins, no hi ha cap suport físic ni cap arxiu que es puguin considerar "monedes". Aquests són associats a les diferents adreces, de la mateixa manera que un banc dóna un valor determinat a un compte bancari.

El procés per generar-los, es coneix amb el nom de "mining". Quan es genera un bloc, s'introdueix una transferència extra en la qual no hi ha adreces d'origen. D'aquesta manera, cada vegada que es crea un bloc es generen nous bitcoins per "recompensar"els miners que han generat el bloc.

Aquest funcionament incentiva als miners a validar els blocs. Com més blocs es verifiquin, més bitcoins es generen, tot i que cada vegada en guanyen menys. El sistema controla que la producció de bitcoins cada vegada es redueixi més. S'ha especificat que la producció de bitcoins mai ha de superar els 21 milions.

S'ha vist que els blocs de transaccions són els fonaments de tot el sistema bitcoin. Hom es pot preguntar, és complicat generar un bloc? Cada quant temps es generen? Per seguretat, no es poden estar generant en segons. Això podria provocar que una transacció invàlida quedés fàcilment tapada per blocs superiors i poder passar desapercebuda fàcilment. El procés que obliga a que els miners inverteixin temps en generar els blocs es coneix amb el nom de "proof-of-work".

Cada bloc conté un número de verificació. Aquest número s'utilitza per generar el hash del bloc. L'objectiu del proof-of-work és trobar un hash que sigui menor a un número determinat. Aquest problema requereix una gran quantitat de còmput, que permet que els miners hagin d'invertir temps en localitzar-lo. L'avantatge d'aquest mètode és que se'n pot ajustar la dificultat. Com més petit hagi de ser el hash, més costarà trobar un número de verificació. El temps de generació s'intenta que sigui d'uns 10 minuts.

Amb les idees donades podem entendre, en general, el funcionament del bitcoin.

Tot i així, és un sistema molt complexe on cada una de les seves parts requeriria un estudi molt profund per poder-ne entendre perfectament els detalls [5]. En el següent apartat es descriurà amb més detall com funciona la generació d'adreces, que és l'objectiu principal d'aquest projecte.

2.3.2 Generació d'adreces

Les adreces bitcoin consten de dues parts fonamentals: una adreça pública que tothom pot veure per a interactuar i una clau privada que permet accedir a l'import que conté l'adreça i fer les transaccions desitjades. El mètode de funcionament és el d'un criptosistema de clau pública. L'algorisme escollit pels bitcoins és l'ECDSA, explicat anteriorment i la corba el·líptica sobre el qual s'aplica l'ECDSA és la SECP256K1.

Per tal de simplificar la notació, a la pràctica s'utilitza una representació de la parella de claus pública i privada per mitjà d'una cadena de caràcters alfanumèrics amb una longitud que oscil·la entre 27 i 34 caràcters.

Pel que fa l'adreça pública, s'obté a partir de la clau pública de l'algorisme d'ECDSA utilitzant els algorismes de hash sha256 i el RIPEMD160, a més de la funció basencode58 per acabar obtenint l'adreça de bitcoins. El procés de generació de l'adreça segueix el següent pseudocodi:

```
[(publicKey_1,publicKey_2),privateKey]=ECDSA();
concat = 04 + publicKey_1 + publicKey_2;
address = sha256(concat);
hash = "00" + RIPEMD160(address);
bytes = sha256(sha256(hash))[0..4];
address = address + bytes;
finalAddress = basencode58(address);
```

El codi funcional s'ha implementat en SAGE i s'adjunta en suport digital.

La clau privada que permetrà l'accés a aquesta adreça, es codifica amb un sistema similar per codificar la clau. Aquest sistema es coneix com Wallet Import Format (WIF). El següent pseudocodi en mostra el funcionament:

```
[(publicKey_1,publicKey_2),privateKey]=ECDSA();
privateKey = "80" + privateKey:
bytes = sha256(sha256(concat))[0..4];
privateKey = privateKey + bytes;
```

```
finalPrivateKey = basencode58(privateKey);
```

Igual que el pseudocodi anterior, s'ajunta el codi implementat en SAGE dins el suport digital.

2.3.3 Wallets

Per poder fer totes les transaccions, controlar les adreces que contenen bitcoins, etc., és necessari un software que ens permeti connectar-nos a la xarxa bitcoin i gestionar les nostres adreces. Aquests softwares es coneixes amb el nom de "Wallets".

Les wallets porten incorporada la generació de claus / adreces i la seva importació. A més, tots ells es connecten a la xarxa bitcoin per poder determinar la quantitat de bitcoins associada a les adreces que tenim dins seu. La majoria d'ells incorpora alguna eina per encriptar les claus generades i així poder-les guardar dins un fitxer, de manera que ens permet fer una importació / exportació de claus més segura. Altres eines que poden incorporar les wallets són generació de codis QR per afavorir l'ús del mòbil en el bitcoin.

Algunes de les wallets més utilitzades són:

- BitcoinQt / Bitcoin Core: es considera el client de referència de la xarxa bitcoin. Implementa tot el protocol i es descarrega totes les cadenes de blocs a l'ordinador. Per aquest motiu, es requereix molt espai disponible al disc dur. A més, implementa poques característiques fora del que és el protocol bitcoin.

- MultiBit: és un client lleuger que es centra en la senzillesa i la velocitat. Es sincronitza amb la xarxa, evitant haver de copiar en local tota la cadena de claus. Dóna suport a molts idiomes i permet l'exportació / importació de claus.

- Hive: és un client lleuger dissenyat per a MacOS X. A part del servei wallet, ofereix altres tipus de serveis com gestió de contactes, còpies de seguretat de les claus (integrades amb Dropbox i TimeMachine) i encriptació de les claus per a la seva protecció.

- Bitcoin Wallet: és un dels clients mòbils més recomanats des del web oficial Bitcoin. Està suportat per Android i BlackBerry. És un dels clients Android que es va veure afectat per la vulnerabilitat de generació de claus.

- Mycelium: és un altre client dissenyat per a Android. Dóna suport a la utilització de codis QR.

- BlockChain: ofereix servei de wallet, però el seu principal atractiu és el portal que han muntat els seus gestors. En ell, amb una interfície molt agradable, es poden consultar un munt de dades sobre les transaccions, el valor dels bitcoins, els últims blocs minats, etc. Cal destacar que tot el seu servei està basat en web. Totes les dades s'emmagatzemen en servidors propis. Per tant, s'ha de confiar en que no se'n farà un mal ús amb elles.

- Electrum: és un client senzill que implementa un tipus de wallet anomenat determinista que s'explicarà al capítol 5 d'aquesta memòria.

- Armory: és un client que també implementa una wallet determinista però d'un tipus diferent a l'Electrum.

Quan escollim una wallet per usar els bitcoins, estem confiant plenament en les persones que han implementat la wallet. L'ús d'una wallet fraudulenta pot provocar que un tercer usuari ens pugui robar fàcilment les claus que la wallet ens ha generat.

Per altra banda, també confiem en que la generació de claus per l'algorisme ECDSA sigui correcta. Com s'ha explicat anteriorment, la clau privada és un nombre aleatori. Si la generació de nombres aleatoris de les wallets no és suficientment bona, podria provocar que fàcilment es generessin parelles de claus repetides. Per tant, si dues persones diferents obtenen un parell de claus iguals, les dues poden accedir als mateixos bitcoins, cosa que trenca tot el sistema muntat. Això és fàcil de comprovar perquè les adreces i el seu saldo són dades públiques en el sistema bitcoin.

A causa d'aquest problema, l'11 d'agost del 2013, es va publicar un llistat [2] de diferents clients per a smartphones Android en els quals s'hi havia detectat una vulnerabilitat en la generació de claus. El problema afectava a aplicacions com Bitcoin Wallet o BitcoinSpinner, wallets que generaven les claus en el propi smartphone sense tenir suficient entropia per tenir bona aleatorietat.

Capítol 3

Anàlisi del codi de generació de claus

La intenció d'aquest capítol és donar un anàlisi del codi de dues wallets encarregat de generar aleatòriament les claus privades utilitzades en el sistema bitcoin. En primer lloc s'analitzarà el codi de la PyWallet. Seguidament s'analitzarà el client oficial i com a conseqüència, el generador de nombres aleatoris que implementa OpenSSL.

3.1 PyWallet

La PyWallet és una eina escrita en llenguatge Python que va néixer com a complement de BitcoinQt per poder exportar a un arxiu de text pla adreces i claus privades a partir d'un arxiu moneder. També en permetia la importació.

Després que Bitcoin-Qt incorporés aquesta funcionalitat a la versió 0.6.0, ja no és necessari l'ús d'aquest complement. Tot i així, PyWallet ha evolucionat fins a implementar una interfície web amb les característiques bàsiques que ha de tenir una wallet convencional, com és la generació de claus privades.

A l'estar escrita en llenguatge Python, el seu codi és molt fàcil d'interpretar per qualsevol persona que tingui coneixements de programació orientada a objectes. Això la fa ideal per a analitzar com genera les claus privades. Per contra, no és una wallet pensada per l'ús d'usuaris comuns, ja que fàcilment es poden deixar claus al descobert i ser fàcilment robades per terceres persones.

3.1.1 Generació de claus privades

Com s'ha vist al capítol de Fonaments teòrics, la generació de claus privades requereix l'ús d'un generador de nombres aleatoris. Seguidament veurem què utilitza la PyWallet per a la generació de les claus privades analitzant el seu codi.

Tota la implementació de la wallet està dins l'arxiu `pywallet.py`. En aquest arxiu es troba definida la classe KEY:

```
class KEY:
    def __init__ (self):
        ...
    def generate (self, secret=None):
        ...
    def set_privkey (self, key):
        ...
    def set_pubkey (self, key):
        ...
    def get_privkey (self):
        ...
    def get_pubkey (self):
        ...
    def sign (self, hash):
        ...
    def verify (self, hash, sig):
        ...
```

Aquesta classe permet gestionar les claus amb les quals treballa la PyWallet. Permet importar-les, a més de generar-les. Observem com implementa el mètode de generació de claus:

```
def generate (self, secret=None):
    if secret:
        exp = int ('0x' + secret.encode('hex'), 16)
        self.prikey = ecdsa.SigningKey.from_secret_exponent (exp,
                      curve=secp256k1)
    else:
        self.prikey = ecdsa.SigningKey.generate (curve=secp256k1)
        self.pubkey = self.prikey.get_verifying_key()
    return self.prikey.to_der()
```

Si no passem cap paràmetre al mètode de generació, ens genera una clau privada amb `ecdsa.SigningKey.generate (curve=secp256k1)`. Aquesta funció es troba a la llibreria `ecdsa` de Python.

Dins la llibreria, al fitxer `keys.py` trobem la següent classe:

```
class SigningKey:
    def __init__(self, _error__please_use_generate=None):
        ...
    def generate(klass, curve=NIST192p, entropy=None,
                 hashfunc=sha1):
        secexp = randrange(curve.order, entropy)
        return klass.from_secret_exponent(secexp, curve, hashfunc)
    def from_secret_exponent(...):
        ...
```

La funció `randrange` és l'encarregada de generar el nombre aleatori per a la clau privada. La funció ve importada de l'arxiu `util.py`. Buscant la funció en aquest arxiu la trobem definida així:

```
def randrange(order, entropy=None):
    if entropy is None:
        entropy = os.urandom
    assert order > 1
    bytes = orderlen(order)
    dont_try_forever = 10000
    while dont_try_forever > 0:
        dont_try_forever -= 1
        candidate = string_to_number(entropy(bytes)) + 1
        if 1 <= candidate < order:
            return candidate
        continue
    raise RuntimeError(...)
```

En cas de no donar-li una font d'entropia, la funció utilitza la llibreria `os` de Python, que conté la funció `urandom`. Aquesta llibreria és oficial de Python i implementa crides del sistema operatiu. Per tant, arribem a la conclusió que la PyWallet, si no li indiquem el contrari, acaba fent una crida al sistema operatiu on s'està executant per a l'obtenció dels nombres aleatoris.

3.2 BitcoinQt

BitcoinQt és considerat el client de referència de la xarxa Bitcoin. Per tant és important assegurar-se que efectua correctament la generació de claus, ja que un error en aquest pot provoca que moltes wallets que l'usen de referència també implementin el mateix error.

3.2.1 Anàlisi del codi

BitcoinQt, a diferència de la PyWallet, està format per un codi font molt més complexe. L'ús de C++ li proporciona més rendiment i optimització mentre que en dificulta la comprensió. Tot i la seva complexitat, se n'ha pogut extreure el generador de nombres aleatoris.

Des de la versió 0.3.23 tot el codi font es troba dins la carpeta `src`. En ella trobem dos fitxers que fan referència a les claus públiques i privades: `key.h` i `key.cpp`. El primer conté les declaracions de mètodes i funcions mentre que el segon les implementa, seguint l'esquema habitual de C / C++.

En el fitxer `key.h` trobem la declaració del mètode `MakeNewKey`, que permet generar una clau privada. La seva implementació és:

```
void CKey::MakeNewKey(bool fCompressedIn)
{
     do {
        RAND_bytes(vch, sizeof(vch));
    } while (!Check(vch));
    fValid = true;
    fCompressed = fCompressedIn;
}
```

Tot i la poca informació que podem extreure d'aquest fragment de codi, és veu clar que la funció `RAND_bytes` és la funció que s'encarrega de generar els bytes aleatoris necessaris per a la generació de la clau privada. Aquests bytes quedaran guardats dins el buffer `vch`.

La funció `RAND_bytes` pertany a la llibreria OpenSSL, de la qual es nodreix BitcoinQt per l'ús de les eines criptogràfiques. El següent pas, serà analitzar el codi de la llibreria OpenSSL per esbrinar si aquesta funció realment és una implementació pròpia d'un generador de nombres aleatoris o si es nodreix d'un altre.

3.2.2 Generació de claus d'OpenSSL

OpenSSL és un projecte open source creat per Eric Young y Tim Hudson que proveeix d'una sèrie d'eines d'administració i llibreries relacionades amb la criptografia. La majoria dels sistemes operatius i molts dels navegadors d'Internet d'avui en dia fan ús d'aquestes llibreries. Bitcoin actualment no n'és l'excepció i també les utilitza.

Tot el codi d'OpenSSL és públic i és totalment accessible mitjançant un repositori Git allotjat als servidors de GitHub. Gràcies això, es facilita molt la tasca de buscar funcions dins el repositori i controlar-ne les versions. En el nostre cas, hem vist que la funció `RAND_bytes` és la que ens genera els bytes de forma aleatòria amb la finalitat de crear una clau privada. Buscant en el repositori, trobem l'arxiu `rand_lib.c` que la implementa. El codi és:

```
int RAND_bytes(unsigned char *buf, int num)
{
    const RAND_METHOD *meth = RAND_get_rand_method();
    if (meth && meth->bytes)
        return meth->bytes(buf,num);
    return(-1);
}
```

Del codi anterior, podem deduir-ne que el mètode `RAND_get_rand_method()` ens retorna un generador de bytes aleatoris el qual invoquem amb la instrucció `meth->bytes(buf,num)`. En el mateix arxiu `rand_lib.c` trobem la implementació d'aquest mètode:

```
const RAND_METHOD *RAND_get_rand_method(void)
{
    if (!default_RAND_meth)
    {
#ifndef OPENSSL_NO_ENGINE
        ENGINE *e = ENGINE_get_default_RAND();
        if(e)
        {
            default_RAND_meth = ENGINE_get_RAND(e);
            if(!default_RAND_meth)
            {
                ENGINE_finish(e);
                e = NULL;
            }
```

```
        }
        if(e)
            funct_ref = e;
        else
#endif
            default_RAND_meth = RAND_SSLeay();
    }
    return default_RAND_meth;
}
```

Com es pot veure, el codi d'aquesta funció no ens diu com es generen els nombres aleatoris, sinó que depenent d'uns certs paràmetres es farà d'una manera o d'una altra. A la documentació d'OpenSSL [8] s'explica que tota la part de mètodes que formen part de la llibreria `ENGINE` el que fan és donar una interfície per a facilitat l'ús d'altres generadors de nombres aleatoris, sobretot els que són generadors hardware. Sabent això, el que queda clar amb el codi anterior és que si no es fa ús de cap generador especial, es crida el generador amb el mètode `RAND_SSLeay()`.

Per veure el codi del mètode `RAND_SSLeay()` hem de canviar a l'arxiu `md_rand.c`. En ell trobem moltes funcions implementades, entre les quals la que busquem. El codi és:

```
RAND_METHOD *RAND_SSLeay(void)
{
    return(&rand_ssleay_meth);
}
```

Aquest mètode senzillament retorna l'adreça d'una estructura inicialitzada a l'inici del mateix arxiu `md_rand.c`:

```
static RAND_METHOD rand_ssleay_meth={
    ssleay_rand_seed,
    ssleay_rand_nopseudo_bytes,
    ssleay_rand_cleanup,
    ssleay_rand_add,
    ssleay_rand_pseudo_bytes,
    ssleay_rand_status
};
```

Arribat en aquest punt, és veu clara la complexitat del codi de les llibreries crip-

togràfiques OpenSSL, ja que no només implementen funcions molt complexes sinó que a més utilitzen propietats avançades dels llenguatges de programació. En aquest cas, ens trobem que es retorna una estructura que s'inicialitza amb diferents funcions criptogràfiques d'un tipus `RAND_METHOD` que no està definit en aquest mateix arxiu.

El fet d'inicialitzar l'estructura amb punters de crides a funció s'explica per la declaració de totes les funcions com a `static`. En llenguatge C, `static` provoca que una funció o objecte només pugui ser usat dins l'arxiu on ha estat definit (similar als mètodes private de la programació orientada a objectes). Per tant, una manera de poder accedir a l'execució d'aquestes funcions és tenir una funció no static que ens retorna una sèrie de punters a les funcions que volem utilitzar fora d'aquest arxiu.

Fins aquí hem vist que OpenSSL ens retorna una estructura de dades de tipus `RAND_METHOD` amb punters a diferents funcions, entre les quals hi ha generació de de nombres aleatoris. Però si tornem a la implementació de `RAND_bytes`, observem que fem la crida `meth->bytes`, que no es correspon al nom de cap de les funcions amb les quals hem inicialitzat l'estructura. Per explicar això, hem de buscar el lloc on s'ha definit l'estructura.

Analitzant l'arxiu `rand.h` trobem la següent definició d'estructura:

```
struct rand_meth_st
{
    int (*seed)(const void *buf, int num);
    int (*bytes)(unsigned char *buf, int num);
    void (*cleanup)(void);
    int (*add)(const void *buf, int num, double entropy);
    int (*pseudorand)(unsigned char *buf, int num);
    int (*status)(void);
};
```

Tot i no coincidir el nom de l'estructura amb el tipus buscat, es veu clarament que és la candidata a ser l'estructura que retorna `RAND_SSLeay`. La segona línia de la definició es correspon a la crida `bytes` que es fa dins el codi de `RAND_bytes` . Però com s'explica el canvi de nom de l'estructura i el tipus de dades `RAND_METHOD`? L'explicació la trobem a l'arxiu `ossl_typ.h`, on tenim una instrucció `typedef` que defineix el tipus de dades:

```
typedef struct rand_meth_st RAND_METHOD;
```

Ara ja quasi hem arribat al final del llarg camí que he seguit dins les llibreries OpenSSL. Pel que hem pogut veure, quan executem la instrucció `meth->bytes` estem cridant la funció `ssleay_rand_nopseudo_bytes` de l'arxiu `md_rand.c`. El codi de la funció és:

```
static int ssleay_rand_nopseudo_bytes(unsigned char *buf,int num)
{
    return ssleay_rand_bytes(buf, num, 0);
}
```

La funció `ssleay_rand_bytes` es troba en el mateix arxiu `md_rand.c` i genera bytes, tant aleatoris com pseudoaleatoris (ho indiquem amb el tercer paràmetre de la crida de la funció). Aquesta funció és la que realment fa tota la generació de bytes aleatoris. No entrarem a analitzar el codi, ja que s'escapa de l'abast d'aquest projecte degut a la seva gran complexitat. Tot i així, el que si podem afirmar és que BitcoinQt usa el generador per defecte que incorpora la llibreria OpenSSL i que està implementat dins la pròpia llibreria sense la necessitat d'usar llibreries externes dels sistemes operatius com fa PyWallet.

El codi de `ssleay_rand_bytes` és extens i no es pretén analitzar-lo. Aquest s'adjunta en suport digital a la memòria en la versió 1.0.1.g d'OpenSSL. Per consultar l'última versió, es pot fer ús del repositori GitHub d'OpenSSL [7].

3.2.3 Versions

Un aspecte important a tenir en compte en l'anàlisi de la generació de claus, és si la forma de generar claus ha canviat des dels inicis dels bitcoins. Gràcies a les eines dels repositoris de control de versions, fàcilment podem veure els canvis fets en aquestes arxius de codi. Uns canvis poc significatius des dels seus inicis donen confiança, ja que almenys per ara, no s'han trobat errors. En canvi, si en determinats moments s'observen canvis molt grans en la generació de claus, això implica que s'ha pogut produir un problema similar al que van patir les wallets que funcionen en Android.

Pel que fa la llibreria OpenSSL, concretament la implementació de funcions random, hi ha control de versions des del 21 de desembre de l'any 1998. Es poden observar pocs commits per la magnitud del projecte OpenSSL. De fet, la majoria dels canvis són per incorporar noves funcionalitats, reorganitzar codi font o control d'errors. No hi ha indicis que s'hagi canviat de manera substancial la generació d'aquests bytes aleatoris. Tot i així, això no implica que possiblement accidentalment s'hagi

provocat algun error per algun canvi menor (com va succeir en el sistema operatiu Debian a l'hora de modificar la versió utilitzada d'OpenSSL).

En el cas del repositori de BitcoinQt [6], tot i ser un projecte de menys vida, és més complicat seguir l'històric de versions. Els diferents merges de branques, una reestructuració de directoris i un canvi d'implementació del codi generador de claus per tal d'optimitzar-ne l'ús provoquen aquesta dificultat. Tot i així, sempre s'ha usat la llibreria OpenSSL per la implementació de l'ECDSA i en concret el seu generador de nombres aleatoris.

Capítol 4

Estudi empíric de l'aleatorietat de les claus

Després d'analitzar la generació de nombres aleatoris que utilitzen diferents wallets de la xarxa bitcoin, hi ha una pregunta òbvia. Realment funciona l'aleatorietat? En aquest capítol es presenten dues comprovacions. La primera consistirà en generar un seguit de claus per comprovar si hi ha bitcoins dins les adreces que ens generen les claus. La segona comprovació és verificar si els generadors de nombres aleatoris trobats en el capítol anterior passen uns tests d'aleatorietat.

4.1 Generació de 100.000 claus privades

Quan estem en una situació on tenim un conjunt d'elements ordenats i volem fer comprovacions sobre ells o buscar-ne algunes propietats, molta gent pensa en començar pel primer i seguir endavant fins a recórrer tots els elements de la llista.

Si el conjunt és petit, habitualment podem recórrer tota la llista. En canvi, si el conjunt és molt gran, habitualment pararem les comprovacions en un determinat moment i s'intentarà una altra estratègia. Però el que és molt probable, és que en algun moment es comenci a recórrer la llista des de l'inici.

Si ens situem en el context dels bitcoins, recordem que tenim un conjunt de nombres que comença des de l'1 fins a l'ordre de la corba el·líptica. Cada nombre serà una clau privada que ens donarà accés a una adreça bitcoin. Una prova molt senzilla que podem implementar (seguint l'argumentació anterior), és agafar nombres des de l'1 fins a n, generar les adreces i mirar si aquestes contenen bitcoins.

En aquest projecte, s'ha elaborat la prova amb els nombres continguts a l'interval $[1, 100.000]$. Per fer-ho s'han implementat 2 scrpts Python. El primer, amb l'ajuda del SAGE, agafa els 100.000 nombres i els transforma en adreces bitcoin. El segon, llegeix aquestes adreces i mitjançant una API del portal blockchain.info, fa peticions GET per rebre dades en format JSON sobre la quantitat de bitcoins que conté l'adreça i les transaccions que s'han fet al llarg del temps.

Si un pensa en la probabilitat de trobar una adreça que contingui bitcoins o que n'hagi tingut en algun moment, al treballar amb punts d'ordre tan elevat, és pràcticament nul·la. Encara que tinguem un interval de 100.000 adreces, aquesta probabilitat segueix sent pràcticament nul·la.

Els resultats d'elaborar aquesta prova no compleixen l'afirmació anterior de les probabilitats. Si bé no s'ha aconseguit trobar cap adreça bitcoin en l'actualitat que contingui alguna quantitat de la moneda, si que se n'han trobat diverses en les quals s'han fet transaccions. A la taula 4.2 en podem veure un resum.

El que destaca més de la taula, és l'adreça generada amb la clau privada 1. S'observen 1089 transaccions. És un nombre increïblement elevat, sobretot tenint en compte que es recomana no reutilitzar mai dues vegades la mateixa adreça Bitcoin. Per aquesta quantitat tan elevada de transaccions, és molt probable que aquesta adreça l'usin molts developers o gent amb coneixements avançats de les wallets per fer proves.

La resta d'adreces bitcoin han tingut entre 2 i 6 transaccions. Són nombres molt més raonables que el cas anterior. Tot i així, en un interval de 100.000 adreces tenim 21 adreces (sense comptar la primera) que han tingut transaccions. És un nombre també massa elevat, tenint en compte que la probabilitat de trobar-ne una sola és pràcticament nul·la.

Mirant les diferents transaccions que s'han fet, es pot veure que moltes d'elles han rebut bitcoins i immediatament s'han transferit a una altra adreça. A partir d'aquí, només es poden fer suposicions sobre el motiu pel qual han tingut bitcoins durant uns segons o minuts. Un podria ser que la persona i/o wallet que utilitza tingués constància que la clau privada conté un nombre molt reduït de dígits i hagués decidit enviar-los a una adreça on la clau privada per accedir-hi sigui més llarga, Però per altra banda, ningú ens assegura que hi hagi persones que hagin fet igual que l'explicat aquí, hagin generat moltes adreces i estiguin esperant que algú i deixi bitcoins.

Una altra fet a destacar, és que algunes de les transaccions d'aquestes adreces tenen una data molt recent. De fet, l'adreça amb clau privada 29.817 durant 4 segons

va contenir 0.001BTC el dia 27/04/2014, data en la qual s'estava elaborant aquest projecte.

Després de realitzar aquesta prova tan senzilla, podem arribar a dues conclusions. Per una banda les adreces a les quals s'hi accedeix amb una clau privada relativament petita són potencialment insegures degut a la facilitat de realitzar proves com la d'aquest projecte. Per altra banda, també queda clar que no tothom genera les claus aleatòriament, tot i que no queden clares les intencions per fer-ho.

Taula 4.1: Adreces Bitcoin amb transaccions

Clau privada	**Adreça**	**Transaccions**
1	1EHNa6Q4Jz2uvNExL497mE43ikXhwF6kZm	1089
2	1LagHJk2FyCV2VzrNHVqg3gYG4TSYwDV4m	3
6	1UCZSVufT1PNimutbPdJUiEyCYSiZAD6n	2
10	1GDWJm5dPj6JTxF68WEVhicAS4gS3pvjo7	2
21	19vxtDbLMNasSpbAEZd7va5Qge6d2zYWbp	2
23	1MNagJqYUsWBD47QbsebCT2Lt7GwZtjkg	4
66	1QAALUJNskZWEbvwKXoTteowgs9xGGeAZR	2
75	1AdaHxALc5jknZVmNtcduvkD33WBv3sX6i	2
91	1P1Dur122Pzey73fFyUgfSbcxDdedSSm4z	2
126	163bgHt747rfMKf7tM6XEoCzhKbvrYgZ6N	2
128	1EoXPE6MzT4EnHvk2Ldj64M2ks2EAcZyH4	2
214	1NvxH7VHMwVdwHCpwLLVhecieHZy3oVPoZ	2
229	1EcDD3QLhSg9zmPnExs5etYaXP5o4yT8xM	6
273	1LDMZk7BE99i8PtUQrnE1jKgwyKipBX4yk	2
613	1FBY6EV1rTrnbQ56EszAUvELLXKNmT7cnq	2
1024	17imJe7o4mpq2MMfZ328evDJQfbt6ShvxA	4
1817	12rdjPMqAuLpqeuMfYPgYQyHFZbW45sbJP	2
1911	1A4JKCgW9Uem8tcYYfDqmHKaizaeMKMjq8	2
1915	1DurkadVnKgU9ANtR1Fm5TtFHdNeoemdMq	2
2304	14tAs3FL2WV1n7beBy1QvKCcigH4NsmpZc	5
14257	1P57hdcbDNnyZcF7vkzgLsJobR3e178UfG	2
29817	19ko28s4PRE5kUcSLqiZTMMYfWMCKqMzfk	2

4.2 Tests d'aleatorietat

Els tests d'aleatorietat són proves estadístiques usades per a decidir si un determinat conjunt de dades es correspon a un patró o pot considerar-se aleatòria. Per a determinar si les claus generades per diferents wallets són realment aleatòries s'han realitzat una sèrie de tests d'aleatorietat.

4.2.1 Els tests del NIST

El NIST (National Institute of Standards and Technology) és una agència del Govern Federal dels Estats Units d'Amèrica. La seva missió és promoure la innovació i la competència industrial dels Estats Units. Entre els diferents àmbits que abasta aquesta agència, promou diferents estàndards d'eines criptogràfiques. Entre aquestes eines hi ha una sèrie de tests d'aleatorietat que l'agència considera necessaris per a considerar que una seqüència sigui aleatòria.

Tots els tests estan detallats en un manual [9] que es pot consultar gratuïtament mitjançant el web del NIST on s'explica el significat de cada test i com interpretar-ne els resultats. La complexitat d'aquests tests i la necessitat d'ús d'eines estadístiques com p-valors o distribucions, fan que una explicació detallada s'escapi de l'abast d'aquest projecte.

4.2.2 Generadors testejats

Recordem que durant l'anàlisi del codi de la PyWallet i el BitcoinQt havíem trobat dos tipus de generadors diferents: els propis del sistema operatiu i el que incorpora OpenSSL. Els tres sistemes operatius més utilitzat avui en dia són Windows, MacOS X i Linux. Per tant a la pràctica, tenim accés a quatre tipus de generadors diferents amb els quals fer les proves.

Sistemes operatius

Per a testejar els generadors dels sistemes operatius s'ha implementat un script en Python que utilitza la llibreria os, igual que fa la PyWallet, per a generar una sèrie de bytes aleatoris. Recordem que aquesta llibreria ens permet executar crides a les llibreries del sistema operatiu.

Cal esmentar que Windows utilitza una llibreria pròpia i que MacOS X i Linux,

al ser de la família dels sistemes Unix, utilitzen urandom. Tot i així, pot haver-hi diferències entre les dues llibreries urandom, ja que Linux no és un derivat de Unix sinó un clon i MacOS X és un derivat de FreeBSD. Per tant és interessant testejar les dues llibreries per separat.

Taula 4.2: Sistemes operatius, màquines i llibreries testejades

Sistema operatiu	**Màquina**	**Llibreria**
Windows 8.1	Torre clònica	CryptGenRandom
MacOS X 10.9	MacBook Pro	urandom
Linux (Ubuntu 14.04)	Portàtil HP Pavilion	urandom

OpenSSL

Per testejar el generador OpenSSL ha estat necessari fer un programa en C per a cridar la funció RAND_bytes. El problema per fer això és que cada sistema operatiu incorpora una versió diferent de OpenSSL. A més, aquesta versió pot haver estat modificada pels desenvolupadors del sistema.

Per a resoldre un possible conflicte amb les versions s'han generat els nombres mitjançant l'última versió disponible d'OpenSSL en el moment d'elaboració del projecte. Aquesta és la versió 1.0.1.g. Per poder utilitzar aquesta llibreria, és necessari obtenir-la dels oficials d'OpenSSL i utilitzar els scripts que incorporen per compilar-la i instal·lar-la al sistema. A l'Apèndix 1 s'especifica com realitzar detalladament aquest procés. També s'adjunten les ordres de compilació del programa que genera els bytes aleatoris.

El programa generador de bytes aleatoris també pot usar sense problemes les llibreries OpenSSL del sistema operatiu, però possiblement s'hagin de fer algunes modificacions. Per exemple, en el cas de MacOS X, Apple no recomana utilitzar aquestes llibreries des de la versió 10.7 de l'operatiu. Per aquest motiu, al compilar salta un avís indicant que les llibreries estan "deprecated". Això es pot solucionar editant les capçaleres del sistema. A més, s'ha de tenir en compte que s'hauran de canviar les ordres de compilació.

4.2.3 Resultats

Pels poder realitzar els tests, s'han generat un total de 4 fitxers (un per cada generador) de mida 1.600.000 bytes. És a dir, en total s'han generat 50.000 claus de 256 bits cadascuna per a cada generador.

La Taula 4.3 mostra els resultats obtinguts[1] pels diferents tests del NIST. En els primers 10 tests es mostra el valor exacte de la proporció de cada test. Per a la mida de seqüències utilitzada es considera que passa el test per a un valor superior a 0.980. En els últims 5 tests es mostra el nombre total de tests que es superen respecte el total de la categoria. Com es pot veure, en totes les fraccions numerador i denominador són el mateix valor indicant que tots els tests es superen.

Per tant, segons els estàndards del NIST qualsevol wallet que utilitzi algun d'aquests quatre generadors de nombres aleatoris es pot considerar apte per a generar claus privades, des del punt de vista de l'aleatorietat.

Taula 4.3: Resultats obtinguts de l'execució dels tests estadístics del NIST

Generador	GNU/Linux	MacOS	Windows 8.1	OpenSSL
Frequency	0.989	0.983	0.992	0.990
BlockFrequency	0.991	0.993	0.986	0.987
Runs	0.984	0.989	0.993	0.993
LongestRun	0.983	0.989	0.983	0.991
Rank	0.994	0.988	0.990	0.996
FFT	0.985	0.987	0.987	0.986
OverlappingTemplate	0.989	0.984	0.988	0.989
Universal	0.992	0.992	0.990	0.989
ApproximateEntropy	0.991	0.989	0.989	0.996
LinearComplexity	0.987	0.992	0.989	0.989
CumulativeSums	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$
NonOverlappingTemplate	$\frac{148}{148}$	$\frac{148}{148}$	$\frac{148}{148}$	$\frac{148}{148}$
RandomExcursions	$\frac{8}{8}$	$\frac{8}{8}$	$\frac{8}{8}$	$\frac{8}{8}$
RandomExcursionsVariant	$\frac{18}{18}$	$\frac{18}{18}$	$\frac{18}{18}$	$\frac{18}{18}$
Serial	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$

[1] L'execució d'aquests tests ha estat realitzada pel Dr. Joan Melià, investigador que en la seva tesi doctoral havia treballat amb aquests tests.

Capítol 5

Deterministic wallets

En els capítols anteriors, s'han tractat wallets que podem classificar de "convencionals": es genera un nombre aleatori, es formen les claus privada i pública i s'incorpora l'adreça associada a la wallet. Si tenim 10.000 adreces, és necessari emmagatzemar 10.000 claus privades. Per resoldre aquest problema d'escalabilitat, s'han creat *Deterministic wallets*.

5.1 Definició i característiques

A diferència de les wallets tradicionals, que generen totes les claus privades de forma aleatòria, les deterministes generen totes les claus privades a partir d'un algorisme específic que utilitza una llavor per ser inicialitzat. Aquesta llavor és inicialitzada de forma aleatòria.

Gràcies a utilitzar aquest sistema, si tenim la llavor d'una wallet determinista i l'escrivim en una altra wallet del mateix tipus, totes les adreces i claus privades de l'anterior wallet es podran visualitzar en la nova. Per tant, és molt fàcil realitzar còpies de seguretat amb aquest sistema, ja que només cal guardar la llavor i no cal preocupar-se de les claus privades de les adreces.

El sistema de les wallets deterministes va més enllà de facilitar la manipulació de les adreces bitcoin. Una característica molt interessant és que proporcionen l'ús d'un parell de claus: una "clau pública mestra" i una "clau privada mestra". La clau pública mestra, generada a partir de la clau privada mestra, permet generar totes les adreces bitcoin de la wallet. Per tant, una persona amb accés a la clau pública mestra pot veure el contingut de totes les adreces bitcoin utilitzades però no pot manipular-ne el contingut, ja que no en té les claus privades. Per generar

aquestes claus privades, és necessari el coneixement de la clau privada mestra. De la mateixa manera, permeten compartir la clau privada de determinades adreces sense comprometre la seguretat de la resta d'adreces.

L'ús del parell de claus mestra és possible gràcies a la següent propietat. Suposem que P és el punt inicial de la corba el·líptica i k_{priv} una clau privada. Per generar la corresponent clau pública, k_{pub}, en l'algorisme ECDSA fem:

$$k_{pub} = k_{priv} \cdot P$$

Si suposem ara que K_{priv} és una clau inicial, la corresponent clau pública serà $K_{pub} = K_{priv} \cdot P$. Ara bé, si i li sumem un determinat enter i qualsevol a K_{priv}, tenim que:

$$k^i_{pub} = (K_{priv} + i) \cdot P = K_{priv} \cdot P + i \cdot P = K_{pub} + i \cdot P$$

Per tant, coneixent només $i \cdot P$ i la clau pública K_{pub} podem conèixer k^i_{pub} sense necessitat de conèixer cap clau privada.

Una altra característica que ofereixen aquest tipus de wallets, en menor o major grau depenent de la wallet, és la jerarquia. Les claus privades que es generen a partir de la clau privada mestra permeten també generar altres claus privades com si elles mateixes fossin claus mestra. Aleshores, això ens permet construir una estructura d'arbre, amb la qual podem fer coses com donar visibilitat o accés a un determinat conjunt d'adreces sense que hi hagi necessitat de donar accés a les altres. A més, donem aquest accés a un conjunt d'adreces donant una sola clau, de forma que cada node intern de l'arbre és en sí una wallet determinista.

Algunes de les wallets més conegudes que implementen aquest sistema són Electrum, CarbonWallet o Armory. No totes utilitzen el mateix sistema de generació de claus com veurem seguidament, tot i que utilitzen la mateixa propietat descrita anteriorment per a generar les claus.

5.2 Tipus de wallets

Actualment hi ha dos models a considerar [3] per aquest tipus de wallets: les basades en Electrum i les especificades en el paper BIP32 [11] i [1]. La tendència és considerar com a estàndard l'especificació del BIP32.

Està previst que Electrum a la versió 2.0.0 implementi el sistema BIP32, ja que

aquest és més robust que el que implementa actualment. Pel que fa les wallets que implementen el BIP32, no totes ho fan en el mateix grau. Per aquest motiu, s'ha de tenir en compte que possiblement no hi hagi compatibilitat entre elles i per tant, la mateixa llavor no acabi generant les mateixes claus.

5.2.1 Electrum

Anem a explicar com es generen les claus en aquesta wallet. En primer lloc, cal escollir una clau privada mestra. Habitualment es dóna una llavor per generar aquesta clau. Sigui K_{priv} la clau generada. El càlcul de la clau pública mestra es genera seguint l'algorisme ECDSA. Per tant, si P és el punt inicial que hem escollit de la corba, la clau pública mestra serà:

$$K_{pub} = K_{priv} \cdot P$$

Una vegada hem calculat les claus mestres, generem un *offset* mitjançant un hash SHA256 que depèn de la clau pública i un índex natural i (el 0 inclòs). És a dir, $C_i = SHA256(k_{pub}, i)$. L'índex ens dóna un ordre en les claus generades. Sigui C_0 el primer offset generat. Per a calcular la primera clau privada que podrem usar, només cal sumar C_0 a la clau privada mestra, de manera que:

$$k^0_{priv} = K_{priv} + C_0$$

Pel procediment habitual, calculem la clau pública:

$$k^0_{pub} = k^0_{priv} \cdot P$$

Observem que aquest sistema, utilitzant la propietat explicada a l'apartat anterior, podem calcular la clau pública sense necessitat d'usar la clau privada:

$$k^0_{pub} = k^0_{priv} \cdot P = (K_{priv} + C_0) \cdot P = K_{priv} \cdot P + C_0 \cdot P = K_{pub} + C_0 \cdot P$$

Com que C_0 depèn únicament de l'índex i la clau pública, es pot calcular fàcilment. Per tant, coneixent únicament K_{pub}, podem calcular k^0_{pub}. Per al càlcul de les successives claus, es repeteix el procés canviant l'índex de l'offset. És a dir, per generar la i-èssima clau privada, calculem l'offset en funció de la clau mestra i el paràmetre i. A la Figura 5.1 en podem veure l'esquema de funcionament. Observem que aquesta wallet només ofereix un nivell de jerarquia. És a dir, les claus generades amb les

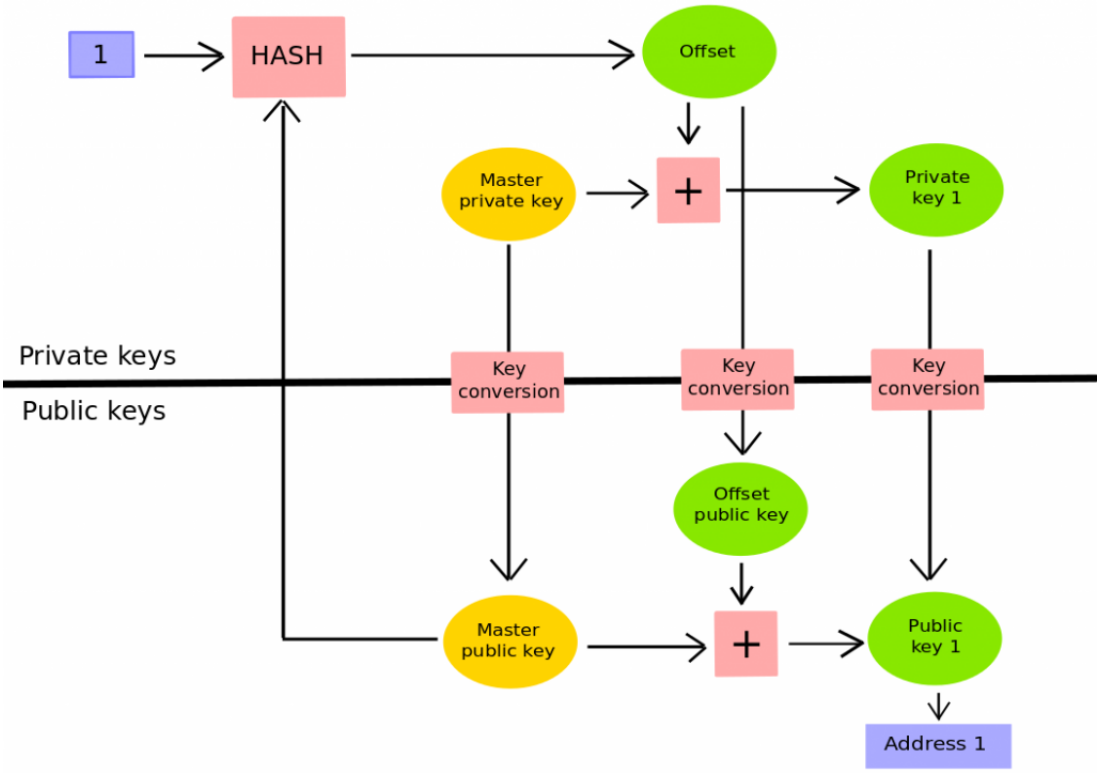


Figura 5.1: Esquema de funcionament d'Electrum

claus mestres no generen altres claus. Per tant implementa parcialment la jerarquia esmentada anteriorment.

Problemàtica del sistema

Tenint en compte que dos dels avantatges de les wallets deterministes són:

- Generar totes les claus públiques només coneixent la clau pública mestra.
- Donar la clau privada d'una adreça a terceres persones sense comprometre la seguretat de la resta d'adreces.

el sistema donat anteriorment té un problema de seguretat. Si un usuari coneix la clau mestra pública K_{pub} i alhora, una clau privada k^i_{priv}, pot adquirir la clau privada mestra. Anem a veure com aconseguir-ho.

Recordem que $k^i_{priv} = K_{priv} + C_i$ i per tant, $K_{priv} = k^i_{priv} - C_i$. Aleshores, si podem conèixer el C_i utilitzat i tenim una clau privada, podem calcular la clau pública mestra. Una manera de fer-ho, tenint en compte la rapidesa del procés, seria per força bruta. C_i només depèn de K_{pub} i l'índex i, només caldria anar generant diferents

claus públiques per a diferents valors de i fins a trobar la nostra. En cas de conèixer l'índex i, ni tan sols és necessari l'ús de la força bruta.

Per tant, arribem a una conclusió: compartir la clau mestra pública per a donar visibilitat del saldo de les adreces i compartir la clau privada d'una d'elles sense comprometre'n la seguretat no és compatible. Si bé el sistema permet fer ambdues coses, com hem vist és insegur fer-les alhora. A més, donat que aquestes wallets només tenen un nivell de jerarquia, un parell de clau mestres només ens permetran usar un sol dels dos avantatges. Aquest és el principal inconvenient de l'ús d'aquestes wallets respecte la implementació que dóna BIP32.

5.2.2 BIP32

BIP32 és una especificació que ens descriu un segon tipus de wallets deterministes. L'especificació contempla la generació d'un arbre de claus. Per tant, a diferència d'Electrum, estem dotant a la wallet d'una estructura jeràrquica de diversos nivells.

La derivació de claus respecte la d'Electrum varia en l'algorisme i els paràmetres d'entrada utilitzats, ja que en el fons es fa el mateix: utilitzar un offset per generar noves claus. Veiem com es realitza una derivació de claus en aquest sistema.

L'algorisme utilitzat en aquest cas és el HMAC-SHA512. Aquest hash es crea en funció de tres paràmetres, tal i com es mostra en la Figura 5.2:

- Una clau pública pare (*Parent Public Key*).
- Una cadena de 256 bits, que depèn de la clau pare (*Parent Chain Code*).
- Un índex de 32 bits (*Index number).*

Observem que els elements que obtenim al final del procés, no només són les claus pública i privada buscades (*Child Public Key*, *Child Private Key*). Alhora estem generant una nova cadena de 256 bits que serà usada en cas de voler derivar de nou aquesta clau (*Child Chain Code*).

Tal i com es mostra en la Figura 5.3, el càlcul de la cadena és senzill. El resultat del hash aplicat, ens retorna un total de 512 bits. Per a generar una clau privada només en necessitem 256. Aleshores, cada hash obtingut es parteix en dues parts de la mateixa mida. Els primers 256 bits seran la nova clau privada. Els 256 restants passen a ser la nova cadena (*Child Chain Code*). En el cas de que les claus de sortida siguin les claus mestres, la cadena serà aleatòria ja que no disposem de cap més ascendent.

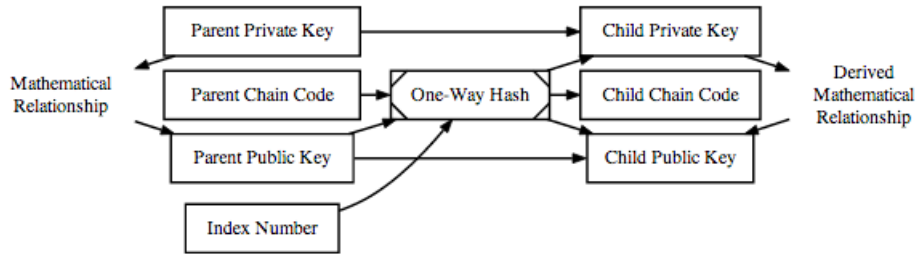


Figura 5.2: Derivació de claus normals BIP32

L'ús d'aquestes cadenes fa que habitualment es parli de claus ampliades o esteses. Una clau ampliada s'entén com una clau privada o pública juntament amb la cadena calculada a l'hora de derivar la clau. Per tant, una clau pública i una clau privada que procedeixin d'un mateix ascendent, tindran la mateixa cadena.

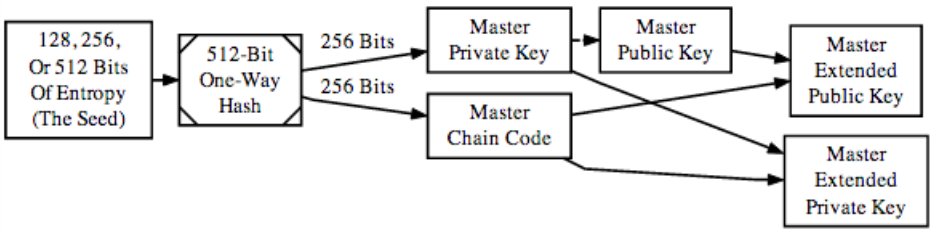


Figura 5.3: Claus ampliades del BIP32

Igual que en el cas de les wallets Electrum, aquesta derivació de claus també ens permet a partir d'una clau pública pare generar-ne els fills sense conèixer-ne les respectives claus privades. A més, al canviar només l'algorisme per calcular l'offset, l'ús d'aquesta forma de derivació de claus manté la problemàtica de les wallets Electrum.

Hardened keys

Fins aquí l'única diferència respecte les wallets Electrum és la jerarquia de claus, ja que amb les cadenes podem generar una estructura d'arbre a partir d'un sol parell de claus mestres. Però BIP32 va més enllà i intenta solucionar el problema de seguretat que té Electrum si es comparteix la clau mestra pública i una clau privada derivada. La solució proposada no arregla totalment el problema, però almenys permet compartir claus públiques i claus privades en determinades condicions.

La solució proposada en el BIP32 és l'ús de les "hardened keys". Aquestes claus eliminen una propietat de les claus normals de manera que a partir d'una clau

pública pare, no es poden generar claus públiques si no es coneixen les claus privades Per aconseguir-ho, el que es fa es canviar la clau pública com a paràmetre d'entrada del hash per la clau privada corresponent, tal com es veu a la Figura 5.4.

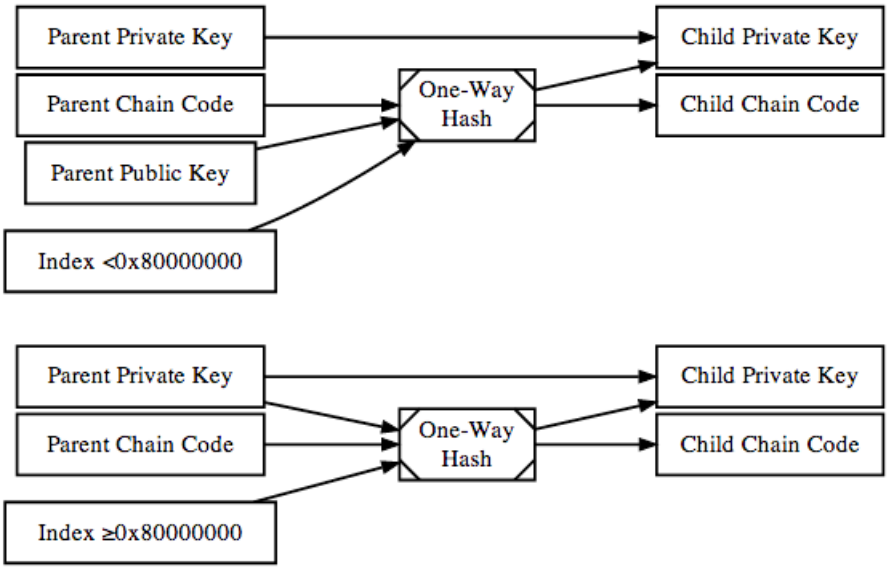


Figura 5.4: Comparació de generació de claus normals i claus hardened

Cal tenir en compte que el BIP32 especifica la quantitat total de claus hardened que es poden crear mitjançant l'índex. Quan l'índex és més gran o igual que 2^{31}, es generaran claus hardened. Si és més petit, es generaran claus pel procediment habitual. Això fa que donat una derivació de claus, podem generar la meitat de claus de cada tipus.

Com hem dit abans, aquesta metodologia de derivació de claus evita el problema de les wallets Electrum. Encara que es conegui la clau pública pare i una clau privada d'algun descendent, no es pot obtenir la clau privada pare. Per contra, elimina una de les principals propietats d'aquestes wallets: no podem donar visibilitat als saldos de diferents adreces donant només la clau pública pare corresponent.

Esquema de derivació de claus

Per esquematitzar l'explicat anteriorment, podem observar les Figures 5.5, 5.6 i 5.7 on s'especifiquen totes les possibles combinacions de derivació de claus. Utilitzarem la següent notació; K_{priv} i K_{pub} són les claus privada i pública respectivament (mestres o no, depenent del node de l'arbre en que estem), k^i_{priv} i k^i_{pub} són les claus privada i pública calculades amb índex i, C és la cadena pare i C_i la cadena obtinguda després de la derivació. El hash el denotarem per la funció f, juntament amb

els subíndex L o R per denotar si considerem la meitat esquerra dels 512 bits o la meitat dreta.

A la Figura 5.5 podem veure la derivació de claus privades normals en forma d'arbre. Associat a aquest arbre hi ha la derivació de claus públiques a la Figura 5.6, on es veuen les dues formes de generar de generar les claus públiques.

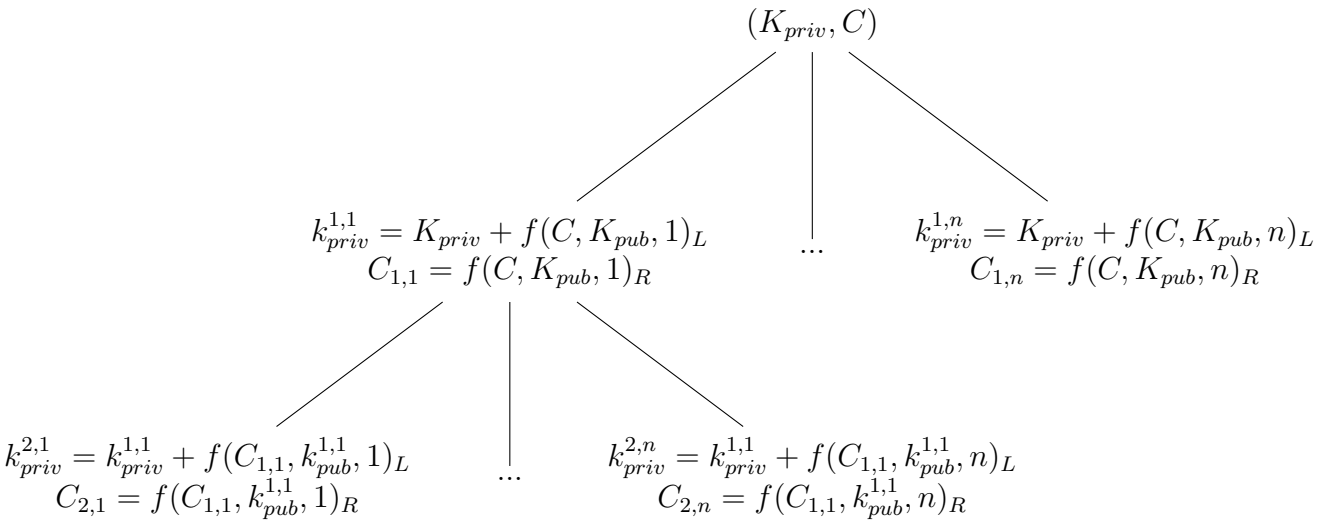


Figura 5.5: Arbre generat per la derivació de claus privades normals

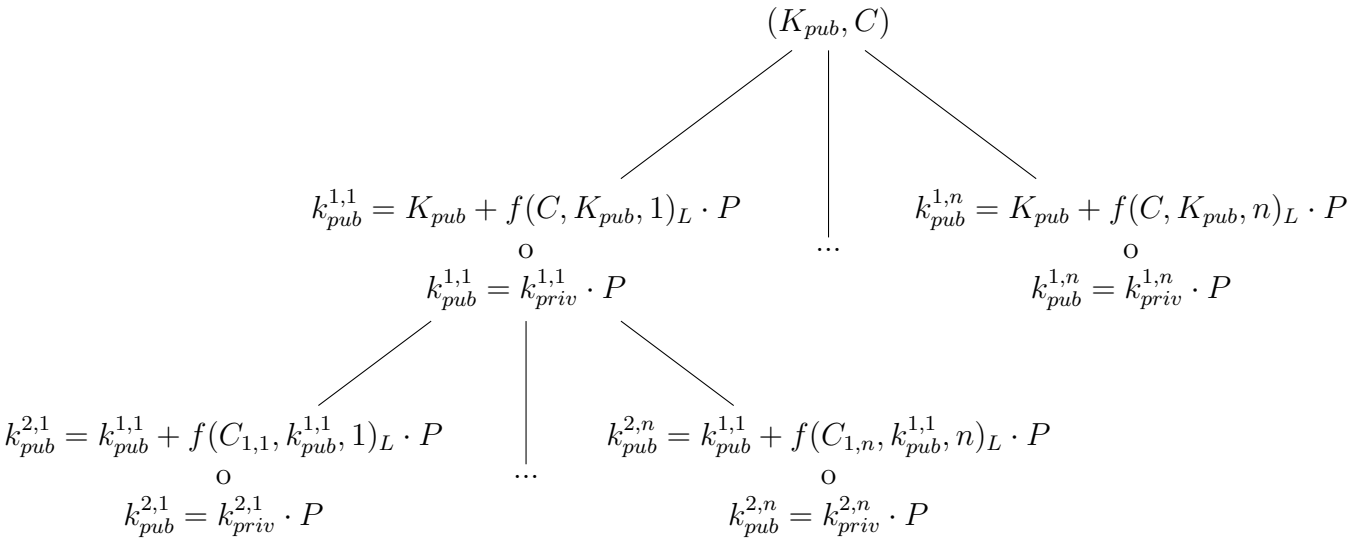


Figura 5.6: Arbre generat per la derivació de claus públiques

En cas de voler derivar claus privades hardened, podem veure l'arbre a la Figura 5.7. Recordem que la diferència entre la derivació normal i la hardened rau en la clau que s'utilitza per calcular les cadenes i que d'una clau hardened podem derivar-ne claus normals.

Cal fixar-se en els índex utilitzats per calcular les claus. Aquests no comencen per l'1 sinó per $n+1$. Recordem que donada una clau privada, podem derivar claus normals i claus hardened, però BIP32 exigeix que l'índex marqui el tipus de clau donada. A la Figura 5.5 és veu que tot node té un total de n claus filles. Per coherència, s'ha continuat amb la mateixa indexació entre les figures i l'índex n marca el final de les claus normals mentre que l'índex $n+1$ marca l'inici de les hardened.

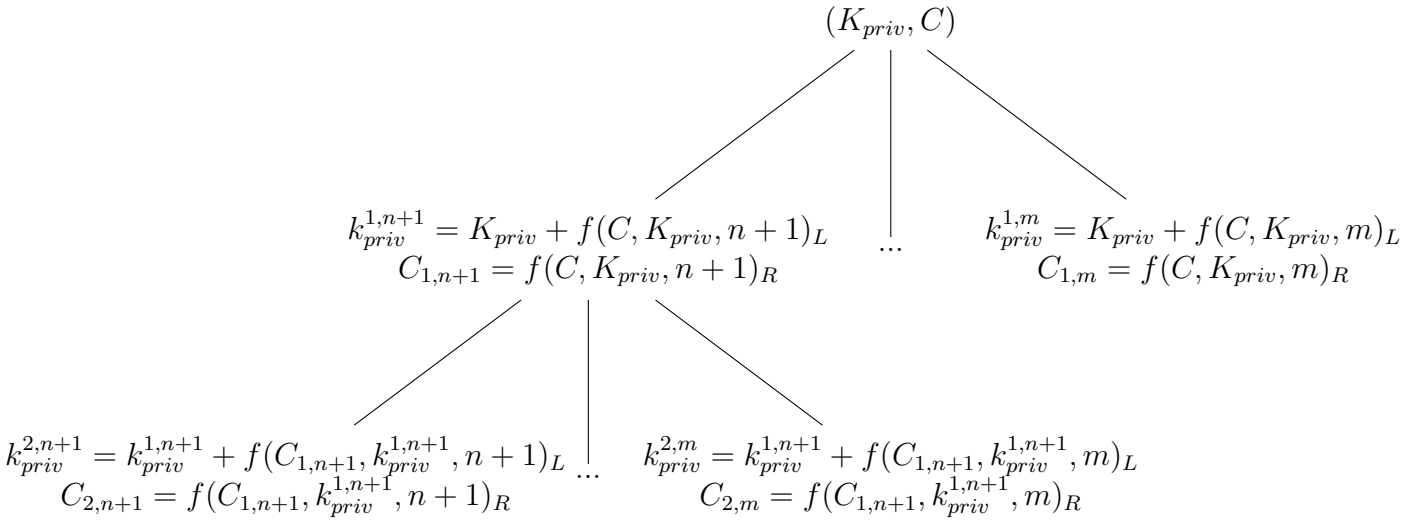


Figura 5.7: Arbre generat per la derivació de claus privades hardened

La derivació de claus públiques que provenen de claus privades hardened només es fa amb el mètode de habitual de generació de claus. És a dir, $k_{pub} = k_{priv} \cdot P$, ja que com hem vist anteriorment, no es poden derivar les claus públiques sense conèixer les clau privada del nivell superior de l'arbre.

Capítol 6

Conclusions

En aquest capítol veurem les conclusions finals del projecte, juntament amb els objectius complerts i les futures línies de treball.

6.1 Conclusions finals

El projecte pretén fer un anàlisi de la generació de claus criptogràfiques que utilitza Bitcoin. Per elaborar-lo s'ha necessitat una base teòrica sobre el funcionament de les corbes el·líptiques i com s'utilitzen dins el món de la criptografia. A més ha estat necessari un estudi previ sobre el funcionament del protocol de bitcoin.

L'anàlisi inicial s'ha centrat en el codi de dues wallets, per poder extreure'n el generador de nombres aleatoris que utilitzen. Aquests generadors són els propis dels sistemes operatius juntament amb el de la llibreria OpenSSL. Per poder provar-los, s'han realitzat els tests d'aleatorietat del NIST, els resultats del qual han estat satisfactoris. Amb aquests resultats, es conclou que no tenim indicis per sospitar de problemes d'aleatorietat en qualsevol wallet que usi algun d'aquests generadors (entenent que no els ha modificat).

Per concloure l'anàlisi de claus criptogràfiques, s'ha introduït el concepte de deterministic wallets, on a partir d'un parell de claus mestres es generen altres claus per poder ser utilitzades en bitcoin. S'han explicat els dos models bàsics des d'un punt de vista més matemàtic i s'ha pogut comprovar que si no se'n fa un ús adequat, permeten que terceres persones puguin obtenir les claus mestres i per tant, accedir a totes les adreces bitcoin que generen aquelles claus.

6.2 Objectius complerts

A l'inici del projecte es van plantejar un total de 5 objectius que s'han anat complint al llarg del projecte. Per complir-los s'ha hagut de variar la línia temporal del projecte, degut a la dificultat de planificar un anàlisi d'aquest tipus.

Els següents punts resumeixen com s'han complert els objectius del projecte:

- S'ha fet un estudi previ sobre el funcionament de les corbes el·líptiques i el seu ús en criptografia. S'ha donat una formalització geomètrica, sense entrar en els detalls matemàtics del pla projectiu per a una millor comprensió del funcionament de les corbes.
- Es dóna un resum sobre el funcionament genèric dels bitcoins fent especial èmfasi en el funcionament de les adreces i les transaccions. No s'han entrat en detalls de mineria ja que s'escapen de l'abast del projecte.
- Al llarg del projecte es pot veure com és d'important que s'utilitzi un bon generador de nombres aleatoris. S'ha presentat un exemple d'un conjunt d'adreces Bitcoin on es veu que l'aleatorietat no ha funcionat correctament, ja que tenim claus d'adreces on s'han produït transaccions.
- Durant l'elaboració del projecte hem vist que les diferents wallets usen llibreries dels sistemes operatius per a ús criptogràfic (com CryptGenRandom o urandom). A més hem vist que el client model utilitza les llibreries externes OpenSSL.
- S'han generat fitxers amb bytes aleatoris amb les diferents llibreries criptogràfiques per a després testejar-los amb la bateria de tests del NIST. Després s'han presentat els resultats del test, que han estat tot aptes.

6.3 Futures línies de treball

El projecte s'ha centrat en l'estudi de les wallets convencionals i l'ús de les llibreries criptogràfiques del sistema operatiu, juntament amb les de l'OpenSSL. A més, s'han introduït el funcionament de les wallets de deterministes i la seva problemàtica. Tenint en compte això, aquests són algunes de les possibles futures línies de treball:

- Analitzar els clients web. Per exemple, bitaddress.org està escrita en JavaScript i utilitza els moviments del mouse com a font d'entropia.

- Analitzar els clients Android. Cal recordar que un dels motius per realitzar aquest projecte és l'error en generació de claus descobert a les wallets del sistema Android.

- Cada vegada sorgeixen més vulnerabilitats a les llibreries OpenSSL i molts són els projectes que decideixen canviar aquestes llibreries per unes altres. Bitcoin no n'és excepció i en breu s'espera que es substitueixi OpenSSL per una llibreria optimitzada a la corba el·líptica SECP256K. Estudiar el funcionament d'aquesta nova llibreria seria una tasca interessant.

- En aquest projecte només s'ha fet una introducció a les deterministic wallets. Es poden analitzar les implementacions que se'n fan i intentar corregir la problemàtica que suposa el seu ús.

Apèndix A

Compilació, instal·lació i execució de OpenSSL

A.1 Compilació i instal·lació de OpenSSL

A.1.1 MacOS X i Linux

En aquests sistemes operatius és necessari compilar les llibreries per a la seva instal·lació. El procés és el mateix en ambdós sistemes excepte en l'execució del primer script.

Les passes a seguir són:

1. Descarregar-se del web de OpenSSL l'última versió de la llibreria.
2. Descomprimir l'arxiu descarregat i obrir un terminal a la ubicació on tenim els fitxers descomprimits amb permisos de superusuari del sistema.
3. Depenent del sistema operatiu executem ordres diferents.

   Linux

   Executar la instrucció

   ```
   ./config
   ```

   Aquesta instrucció també es pot executar a Mac OS X, però per problemes d'implementació no detecta correctament el sistema operatiu i falla.

MacOS X

Executar la instrucció

```
./Configure darwin64-x86_64-cc
```

Aquesta és més potent que l'anterior ja que permet compilar les llibreries en qualsevol sistema operatiu passant-li els paràmetres correctes.

4. Executar les següents instruccions, una rere l'altra.

```
make
make test
make install
```

5. Les llibreries ja estan compilades i instal·lades al sistema per poder-les utilitzar. Cal notar que la instal·lació no substitueix la versió de les llibreries OpenSSL que porta el sistema operatiu per defecte. Per tant, si comprovem la versió de OpenSSL al terminal o intentem compilar un programa amb els paths del sistema per defecte, estarem utilitzant les llibreries del sistema, no l'última versió instal·lada.

A.1.2 Windows

Per a la instal·lació de l'última versió de les llibreries OpenSSL en aquest sistema operatiu, només cal descarregar-se dels repositoris oficials l'última versió dels arxius binaris precompilats del web d'OpenSSL.

La llibreria s'ofereix en dues versions, una per l'arquitectura de 32 bits i l'altra per l'arquitectura de 64 bits. Independentment de la versió de Windows utilitzada, es recomana l'ús del precompilat de 32 bits.

La instal·lació només consisteix en seguir les passes de l'assistent d'instal·lació. La configuració per defecte és la recomanada.

A.2 Compilació i execució d'arxius

L'explicació que es fa a continuació és vàlida per a qualsevol sistema derivat de Unix. En el cas de Windows, s'han d'incloure els paths del directori on s'ha instal·lat OpenSSL al compilador.

Suposant que s'han seguit els passos anteriors, els directoris per defecte on s'instal·la OpenSSL és:

```
usr/local/ssl/lib
usr/local/ssl/include
```

El primer conté les llibreries i el segon les capçaleres. En cas de voler compilar usant aquestes llibreries i el compilador `gcc`, s'ha d'executar el següent format d'ordre:

```
gcc <nom arxiu a compilar.c> -o <nom de l'executable>
-L/usr/local/ssl/lib -I/usr/local/ssl/include
-l<nom de la llibreria> -m<32 o 64>
```

Si es vol incloure més d'una llibreria, nomes cal incloure els noms amb el `-l` al davant per cada llibreria.

Per últim, és molt important especificar la instrucció correctament. Aquesta fa referència a l'arquitectura del sistema operatiu[1]. Si usem un sistema de 32 bits, la instrucció serà `-m32`. En cas de sistemes de 64 bits, `-m64`. Això és necessari perquè OpenSSL s'haurà compilat en l'arquitectura del sistema operatiu.

Un exemple de compilació seria el següent (compila el fitxer de generació de nombres aleatoris usat en el projecte):

```
gcc rand.c -o bytes -L/usr/local/ssl/lib -I/usr/local/ssl/include
-lcrypto -m64
```

[1] Avui en dia, quasi tot el hardware funciona amb 64 bits. Tot i així, és possible que el sistema operatiu instal·lat sigui de 32 bits.

Bibliografia

[1] bitcoin.org. Developer guide. https://bitcoin.org/en/developer-guide. Última consulta 18-06-2014.

[2] bitcoin.org. Android security vulnerability. http://bitcoin.org/en/alert/2013-08-11-android, Agost 2013. Última consulta 18-06-2014.

[3] Vitalik Buterin. Deterministic wallets, their advantages and their understated flaws. http://bitcoinmagazine.com/8396/deterministic-wallets-advantages-flaw/. Última consulta 18-06-2014.

[4] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *The Cryptography Mailing List*, 2008. Disponible a: http://bitcoin.org/bitcoin.pdf . Última consulta 18-06-2014.

[5] Varis. Bitcoin wiki. https://en.bitcoin.it/wiki/Main_Page. Última consulta 18-06-2014.

[6] Varis. Github bitcoin. https://github.com/bitcoin/bitcoin. Última consulta 18-06-2014.

[7] Varis. Github openssl. https://github.com/openssl/openssl/. Última consulta 18-06-2014.

[8] Varis. Openssl wiki. http://en.wikibooks.org/wiki/OpenSSL. Última consulta 18-06-2014.

[9] Varis. A statistical test suite for random and pseudorandom number generators for cryptographic applications. http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf, Abril 2010. Última consulta 18-06-2014.

[10] Kevin Wilson. Introduction to groups and elliptic curves. `https://web.math.princeton.edu/~khwilson/toorcamp/group_intro.pdf`, Juliol 2009. Última consulta 18-06-2014.

[11] Pieter Wuille. Bip 0032. `https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki`. Última consulta 18-06-2014.

Firmat: Josep Miquel Andreu Alemany
Bellaterra, juny de 2014

Resum

El present projecte realitza un anàlisi de les claus criptogràfiques utilitzades en bitcoin. El projecte introdueix les nocions bàsiques necessàries de les corbes el·líptiques, la criptografia de corbes el·líptiques i els bitcoins per a realitzar l'anàlisi. Aquest anàlisi consisteix en explorar el codi de diferents wallets bitcoin i realitzar un estudi empíric de l'aleatorietat de les claus. Per últim, el projecte introdueix el concepte de wallet determinista, el seu funcionament i alguns dels problemes que presenta.

Resumen

El presente proyecto realiza un análisis de las claves criptográficas utilizadas en bitcoin. El proyecto introduce las nociones básicas necesarias de las curvas elípticas, la criptografía de curvas elípticas y los bitcoins para realizar el análisis. Este análisis consiste en explorar el código de diferentes wallets bitcoin y realizar un estudio empírico de la aleatoriedad de las claves. Por último, el proyecto introduce el concepto de wallet determinista, su funcionamiento y algunos de los problemas que presenta.

Abstract

This project performs an analysis of cryptographic keys used in Bitcoin. The project introduces the necessary basic knowledge of elliptic curves, elliptic curve cryptography and bitcoins for analysis. This analysis wants to explore the code of different Bitcoin wallets and perform an empirical study of the randomness of the key. Finally, the project introduces the concept of deterministic wallet, its functioning and some of the problems that it presents.